Towards Compiling SAC for the Xeon Phi Knights Corner and Knights Landing Architectures — Strategies and Experiments —

Clemens Grelck
University of Amsterdam
Amsterdam, Netherlands
C.Grelck@uva.nl

Nikolaos Sarris
University of Amsterdam
Amsterdam, Netherlands
Nikolaos.Sarris@student.uva.nl

ABSTRACT

Xeon Phi is the common brand name of Intel's Many Integrated Core (MIC) architecture. The first commercially available generation *Knights Corner* and the second generation *Knights Landing* form a middle ground between modestly parallel desktop and standard server processor architectures and the massively parallel GPGPU architectures.

In this paper we explore various compilation strategies for the purely functional data-parallel array language SaC (Single Assignment C) to support both MIC architectures in the presence of entirely resource- and target-agnostic source code. Our particular interest lies in doing so with limited, or entirely without, user knowledge about the target architecture. We report on a series of experiments involving two classical benchmarks, Matrix Multiplication and Gaussian Blur, that demonstrate the level of performance that can be expected from compilation of abstract, purely functional source code to the Xeon Phi family of architectures.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**;

KEYWORDS

Array processing, Single Assignment C, multithreading, automatic parallelisation, Xeon Phi

ACM Reference Format:
Clemens Grelck and Nikolaos Sarris. 2017. Towards Compiling SAC for the Xeon Phi Knights Corner and Knights Landing Architectures — Strategies and Experiments —. In *IFL 2017: 29th Symposium on the Implementation and Application of Functional Programming Languages, August 30-September 1, 2017, Bristol, United Kingdom,* Nicholas Wu (Ed.). ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3205368.3205377

IFL 2017, August 30-September 1, 2017, Bristol, United Kingdom

ACM ISBN 978-1-4503-6343-3/17/08…$15.00
https://doi.org/10.1145/3205368.3205377

1 INTRODUCTION

SaC (Single Assignment C) is a purely functional, data-parallel array language [13, 16, 17] with a C-like syntax (hence the name). SaC features homogeneous, multi-dimensional, immutable arrays and supports both shape- and rank-generic programming: SaC functions may not only abstract from the concrete shapes of argument and result arrays, but even from their ranks (i.e. the number of dimensions). A key motivation for functional array programming is fully compiler-directed parallelization for various architectures starting from exactly the same one source. Currently, the SaC compiler supports general-purpose multi-processor and multi-core systems [12], CUDA-enabled GPGPUs [18], heterogeneous combinations thereof [7], the Amsterdam MicroGrid general-purpose many-core processor [14] or, most recently, clusters of workstations [21].

Xeon Phi is the common brand name of Intel's Many Integrated Core (MIC) architecture. The MIC architecture essentially bridges the gap between modestly parallel standard desktop and server processor architectures, namely Intel's own Xeon product line, and the highly successful domain of massively parallel general-purpose graphics processing units (GPGPUs) dominated by NVidia. Like GPGPUs the MIC architecture trades sequential performance in favour of the tight integration of a large number of compute cores. Thus, parallel execution is the mandatory default rather than an add-on.

Very much unlike GPGPUs, the Xeon Phi cores are universal compute cores, capable of running an adapted, but otherwise fully-fledged Linux operating system. Consequently, we can make use of the standard parallel programming abstractions from shared memory models like OpenMP and Posix Threads to distributed memory models like MPI. Indirectly any higher level parallel programming abstraction that is built on top of them likewise benefits from the Xeon Phi family of architectures. From a software engineering point of view this is arguably the most relevant advantage of the Xeon Phi over GPGPUs, which require a significantly more disruptive programming and performance engineering approach.

Intel released the first commercially available generation, coined *Knights Corner*, in November 2012 [4]. It has 57-61 cores running at 1053MHz; each core is capable of executing 4 threads (pseudo-)simultaneously. The peak performance is

advertised as 1011 GFLOPS. Like GPGPUs Knights Corner comes as an extension board for the PCIe bus.

Intel released the second and currently latest Xeon Phi generation, named *Knights Landing*, in June 2016 [1]. It has 64–72 cores running at 1300–1500 MHz and like its predecessor it can run 4 threads simultaneously. Depending on the exact model Intel reports the peak performance as between 2662 GFLOPS and 3456 GFLOPS. Unlike Knights Corner, the current generation Knights Landing is a true host processor architecture and thus avoids the PCIe bus altogether. Both Xeon Phi generations owe much of their nominal peak performance to 512-bit vector processing units, able to execute 16 single-precision or 8 double-precision operations per cycle.

The specific contribution of this paper is to explore three compilation strategies for SaC for both Xeon Phi generations and to report on preliminary experimental evaluation of the actual performance achieved. We consider this timely and relevant because the processor landscape is quickly evolving, parallel architectures are the clear trend, and the persistent question is how functional languages, in particular those geared towards parallel computing, and their implementations cope with the continuously evolving hardware challenges.

At the same time, our work adds one more highly relevant parallel computer architecture to the portfolio of SaC. We come yet another step closer to our overarching goal of supporting parallel computing, regardless of the concrete execution machinery, from a single architecture- and resource-agnostic functional specification. The preliminary experiments demonstrate that substantial performance levels can be achieved "for free", at least from the user's perspective. The technical peak performance, however, still leaves a major gap, as could be expected.

The remainder of the paper is organised as follows. Section 2 provides some background information on our functional array language SaC while Section 3 introduces the Xeon Phi architecture, namely Knight's Corner and Knight's Landing in more detail. In Section 4 we sketch out the various strategies towards compiling SaC to the Xeon Phi that we investigated. Our preliminary experimental evaluation is discussed in Section 5. Finally, we sketch out related work in Section 6 before we draw conclusions in Section 7.

2 INTRODUCING SAC

As the name "Single Assignment C" suggests, SaC leaves the beaten track of functional languages with respect to syntax and adopts a C-like notation. This is meant to facilitate adoption in compute-intensive application domains, where imperative concepts prevail. Core SaC is a functional, side-effect free variant of C: we interpret assignment sequences as nested let-expressions, branches as conditional expressions and loops as syntactic sugar for tail-end recursive functions; details can be found in [16].

Despite the radically different underlying execution model (context-free substitution of expressions vs. step-wise manipulation of global state), all language constructs adopted from C show exactly the same operational behaviour as expected by imperative programmers. This allows programmers to choose their favourite interpretation of SaC code while the compiler exploits the benefits of a side-effect free semantics for advanced optimisation and automatic parallelisation.

2.1 Array Programming

On top of the scalar language kernel SaC provides genuine support for truly multidimensional and truly stateless/functional arrays. Arrays may be passed between functions without restrictions: they are truly first class citizens.

Array types include arrays of fixed shape, e.g. `int[3,7]`, arrays of fixed rank, e.g. `int[.,.]`, and arrays of any rank, e.g. `int[*]`. The latter include scalars, which we consider rank-0 arrays with an empty shape vector. For convenience and equivalence with C we use `int` rather than the equivalent `int[]` as a type notation for scalars.

The hierarchy of array types induces a subtype relationship, and SaC supports function overloading with respect to subtyping. SaC only features a very small set of built-in array operations, namely to query the rank or shape of arrays or to index into arrays. In contrast, all aggregate array operations are specified using WITH-loop array comprehensions:

```
with {
  ( lower_bound <= idxvec < upper_bound) : expr;
   ...
  ( lower_bound <= idxvec < upper_bound) : expr;
}: genarray( shape, default)
```

The keyword `genarray` characterises the WITH-loop as an array comprehension that defines an array of shape *shape*. The default element value is *default*, but we may deviate from this default by defining one or more index partitions between the keywords `with` and `genarray`.

Here, *lower_bound* and *upper_bound* denote expressions that must evaluate to integer vectors of equal length. They define a rectangular (generally multidimensional) index set. The identifier *idxvec* represents elements of this set, similar to loop variables in FOR-loops. Unlike FOR-loops, we deliberately do not define any order on these index sets. We call the specification of such an index set a *generator* and associate it with some potentially complex SaC expression that is in the scope of *idxvec* and thus may access the current index location. As an example, consider the WITH-loop

```
A = with {
      ([1,1]<=iv<[4,5]): 10*iv[0]+iv[1];
      ([4,0]<=iv<[5,5]): 42;
    }: genarray( [5,5], 99);
```

that defines the 5×5 matrix

$$A = \begin{pmatrix} 99 & 99 & 99 & 99 & 99 \\ 99 & 11 & 12 & 13 & 14 \\ 99 & 21 & 22 & 23 & 24 \\ 99 & 31 & 32 & 33 & 34 \\ 42 & 42 & 42 & 42 & 42 \end{pmatrix}$$

WITH-loops in SaC are extremely versatile. In addition to the dense rectangular index partitions shown above SaC

supports also strided generators. In addition to the `genarray`-variant used here, SAC features further variants, among others for reduction operations. Furthermore, a single WITH-loop may define multiple arrays or combine multiple array comprehensions with further reduction operations, etc. For a complete, tutorial-style introduction to SAC as a programming language we refer the interested reader to [13].

2.2 Multithreaded Execution

Compiling SAC programs into efficiently executable code for a variety of parallel architectures is a non-trivial undertaking. While this clearly is not the place to explain the compilation process in any detail, we still sketch out a few areas most relevant for our current work.

It probably comes at no surprise to the experienced reader that WITH-loops, as introduced in the previous section, play a central role in the compilation of SAC programs. Any aggregate array operation in SAC, in one way or another, is expressed by means of WITH-loops, may it be explicitly in the application program or implicitly through composition of basic operations from the SAC standard array library. Thus, we can expect that almost any SAC program spends almost all execution time in WITH-loops.

Many of our optimisations are geared towards the composition of multiple WITH-loops into one [15]. These compiler transformations systematically improve the ratio between productive computing and organisational overhead. Consequently, when it comes to generating code for parallel execution on pretty much any architecture, we can focus on WITH-loops. WITH-loops are data-parallel by design. Thus, any WITH-loop can be executed in parallel without further analysis.

Our code generator for multi-/many-core standard server processor architectures follows an offload model (or fork/join-model), as illustrated in Fig. 1. Program execution always starts in single-threaded mode and only if the execution reaches a WITH-loop, worker threads are created that join the master thread in the execution of the data-parallel WITH-loop. A WITH-loop-scheduler assigns array elements for computing to among the worker threads according to one of several policies that range from static scheduling to dynamic self-scheduling with and without affinity control. When no more work is available, the worker threads terminate and, having waited for the last worker thread, the master thread resumes single-threaded execution. For the time being we refrain from recursice unfolding of parallel execution and run WITH-loops nested inside other WITH-loops sequentially. A simple heuristic avoids parallel execution that is suspected to create more overhead than benefit.

The total number of threads, eight in the illustration of Fig. 1, is once determined at program startup and remains the same throughout program execution. This number is typically motivated by the hardware resources of the deployment system. Due to the malleability property of the data-parallel applications concerned, application characteristics are mostly irrelevant. While it would be technically simple to determine

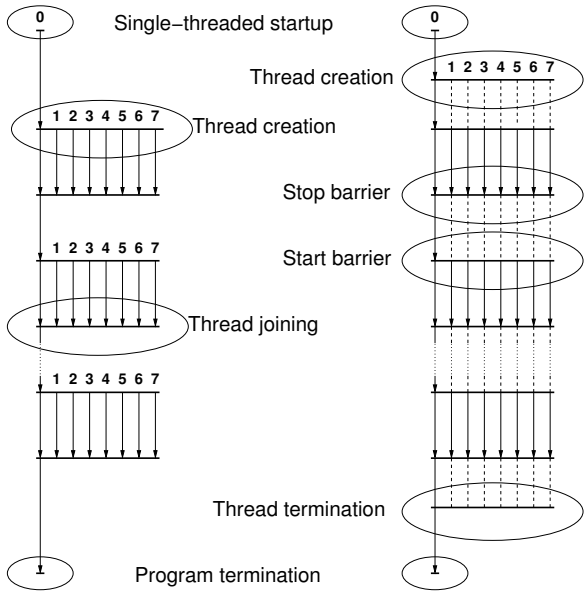

Figure 1: Multithreaded execution models: conceptual fork-join model (left) and start/stop barrier implementation (right)

the number of available cores at application start, SAC for the time being expects this number to be provided by the user, either through a command line parameter or through an environment variable.

As illustrated on the right hand side of Fig. 1 SAC does not literally implement the fork/join-model, but rather starts the worker threads right at the beginning and before the first WITH-loop is encountered during program execution. All worker threads are preserved until program termination. The conceptual fork/join model is implemented through two dedicated barriers: the *start barrier* and the *stop barrier*. At the start barrier worker threads wait for activation by the master thread. At the stop barrier the master thread waits for all worker threads to complete the parallel section while the worker threads immediately pass on to the following start barrier. We use highly efficient tailor-made implementations that exploit properties of the cache coherence protocol, but are essentially based on spinning.

More information on SAC's multithreaded code generation backend can be found in [11, 12]. The level of detail used above is motivated by the fact that this background work forms the basis for supporting both the Knights Corner and the Knights Landing architectures in our current work.

3 XEON PHI ARCHITECTURE

The Knights Corner and the Knights Landing processor architectures are the beginning of a new product line at Intel, but in fact they continue a line of development that started much earlier. From the beginning the motivation has been to address the ever-growing success of GPGPUs while retaining the x86 instruction set architecture as far as possible.

3.1 From Larrabee to Knights Ferry

With the *Larrabee* microarchitecture (2006–2010) Intel targeted highly parallel visual computing applications [23]. However, the attempt to directly compete with NVidia's GPGPUs apparently failed, and the Larrabee project was terminated before the release of a commercial product.

In 2009 Intel prototyped the Single-chip Cloud Computer (SCC) [22]. The SCC features 48 P54C cores in 24 tiles of two cores each. The cores communicate via a 2d-mesh network, hence the analogy to cloud computing. One strength of the SCC was the ability to adjust the clock frequency per tile and the voltage per voltage island of 8 cores. Lack of cache coherence, however, made programming the SCC challenging, and the absence of tooling made the SCC never rise beyond academic research.

In 2010 Intel announced the first prototype of the novel *Many Integrated Core* Architecture (MIC), named *Knights Ferry*. The Knights Ferry architecture featured 32 cores, had 2 GB of memory and its single board peak performance exceeded 750 GFlops.

3.2 Knights Corner

Next in line is Intel's first commercial many-core product: *Knights Corner (KNC)* [4, 19], our first target architecture. Knights Corner comes in the form of a coprocessor that is connected with a host system via the PCIe bus. A virtual TCP/IP stack is implemented over the bus to provide communication between the host and the coprocessor as a network node. Accordingly, users are able to connect to the chip through a secure shell (ssh) and start their applications on the Xeon Phi directly.

Furthermore, users are able to build heterogeneous applications, parts of which execute on the host and parts on the coprocessor. Taking this extra step in program development could be well justified as the sequential performance of the Knights Corner architecture, among others due to a short in-order pipeline, is far from competitive with contemporary general-purpose Intel Xeon processors. Hence, the Knights Corner is a classical accelerator architecture for compute-intensive, throughput-oriented applications.

The Knights Corner architecture features between 57 and 61 cores. Each core is 4-fold hardware-threaded, i.e. a Knights Corner system can support between 228 and 244 hardware threads. Our system used in the experiments described in Section 5 has 60 cores and hence 240 hardware threads.

Each core has 32 KB L1 instruction cache, 32 KB L1 data cache and 512 KB unified L2 cache memory. All caches are kept coherent with a global, distributed tag directory. All cores are connected via a bidirectional ring interconnect. The memory controllers and the PCIe client logic provide the interface to the GDDR5 memory on the coprocessor and the PCIe bus, respectively.

A key component of the architecture is the Vector Processing Unit (VPU) included in each core. The VPU features a 512-bit SIMD instruction set and, hence, is able to execute 16 single-precision or 8 double-precision operations per cycle. Using Fused Multiply-Add (FMA) instructions instruction-level parallelism even grows to 32 single-precision or 16 double-precision operations at once.

The bidirectional ring interconnect, has three independent rings for each direction. The *data block ring* is 64 Bytes wide to support high bandwidth load operations. The *address ring* is used to send read/write commands and memory addresses. Finally, the *acknowledgement ring* is the smallest one and is used to send flow-control messages.

The cache-miss mechanism is quite simple. In the case of an L2 cache miss, an address request is sent through the address ring to the tag directories. If the requested block is found in another CPU's cache, a forwarding request is sent to this cache, and subsequently the requested block is sent through the data ring. If the requested block is not found in any cache, the address is sent to the memory controller.

3.3 Knights Landing

Intel revealed the second generation Xeon Phi architecture, named *Knights Landing (KNL)*, at the International Supercomputing Conference (ISC) in June 2016 at Frankfurt (Main), Germany [20, 25]. The greatest difference between KNC and KNL is that KNL is a standalone processor, liberated from memory limitations and PCIe transfers between host and device memory. KNL also uses a more power efficient core design derived from Intel's Atom processor. It featurea a two-wide, out-of-order pipeline to narrow down the gap between Xeon Phi and general-purpose processors in sequential execution. According to Intel the Knights Landing architecture delivers over 3 TFLOPS double-precision peak performance while providing 3.5 times higher performance per Watt than the KNC.

KNL's Instruction Set Architecture is binary compatible with prior Intel processors. Peak performance is expected to be achieved from code that uses the AVX-512 ISA that provides access to 512-bit vector units already characteristic for the Knights Corner architecture, but now with an extended instruction set.

The KNL processor consists of 72 cores and 144 vector processor units (VPU) connected by a 2d-mesh interconnect, and just as the KNC the KNL supports four hardware threads per core. The processor's basic building block is the *tile* which consists of two cores, two VPUs per core and 1 MB L2 cache memory shared across the two cores. Each core is able to access the L2 caches of any other tile as well as the memory through the interconnect which implements a MESIF cache-coherence protocol whose states are stored in a distributed tag directory.

The KNL is shipped with two types of memory: 16GB Multi-Channel DRAM (MCDRAM) designed for high bandwidth up to 450 GB/s and DDR4 RAM designed for capacity. Two memory controllers with three channels each support up to 384GB of memory at an aggregate bandwidth of 90 GB/s. Furthermore, programmers are able to select how they want to use MCDRAM by choosing one of multiple configurations. The first available configuration is the *flat mode* which treats

MCDRAM as regular memory and hence the total memory is seen as a NUMA system with two nodes (MCDRAM and DDR4). The second option is the *cache mode* that operates the entire MCDRAM as cache for the DDR4. This is the default configuration option. The third option is called *hybrid mode* and with this 25% or 50% of the MCDRAM can be used as cache. The percentage of MCDRAM used as cache can be chosen in the BIOS.

3.4 Future Developments

Until November 2017 the Xeon Phi product line seemed to be well on track for the future. Intel had announced the third generation MIC architecture: *Knights Hill* [10]. The US Department of Energy planned to include Knights Hill processors in their new Aurora supercomputer. Furthermore, Intel had announced a Xeon Phi version specialised for Deep Learning, named *Knights Mill* [24].

However, at SC'17 Intel announced that the Knights Hill architecture would be replaced by a new microarchitecture specifically designed for exascale computing [6, 9]. How this new platform will look like is still unknown, and concrete results are not expected before the early 2020s. It seems plausible to expect the existing Xeon Phi architectures to have a lasting impact on future microarchitecture designs, nonetheless.

4 COMPILATION STRATEGIES

GPGPUs are notoriously sensitive to control flow deviations among threads: whenever one thread takes a different route than another thread, system performance quickly degrades. Hence, GPGPUs are only the architecture of choice for application components that more or less strictly obey the single control flow multiple data principle. In contrast, the big advantage of the Xeon Phi family is to offer a much more generic acceleration option. As already mentioned, both KNC and KNL are able to run a fully-fledged Linux kernel. Accordingly, this model provides the opportunity to run entire programs on the KNC in the same way as these programs could run on a general-purpose x86 processor. In the following we discuss three different approaches for running SAC on the Xeon Phi:

- The *native model* assumes that the user explicitly logs into the Xeon Phi system, typical by means of `ssh`, and runs the code on the system. The disadvantage of this approach is that it somewhat runs counter the SAC ideal of keeping parallel execution as much as possible under the hood.
- The *uniform offload model* addresses this issue and allows users to initiate program runs from the general-purpose host system. The compiled code itself offloads the actual computation to the Xeon Phi, largely transparent to the user.
- The *selective offload model* aims at running only data-parallel kernels on the Xeon Phi and to keep the remaining sequential computations on the host processor. This approach largely follows our strategy for supporting GPGPUs [18].

All three approaches essentially rely on SAC's existing compilation path for multithreaded execution [12]. The Xeon Phi (co)processor family creates another veritable stress test for its design and implementation.

4.1 Native Model

The main challenge imposed by the native model is to cross-compile code for the specific instruction sets of the KNC coprocessor and the KNL processor, among others to make use of their 512-bit vector processing units. Fortunately, the SAC compiler is highly customisable with respect to the needs of diverse target architectures including varying backend C compilers.

Different versions of compiled SAC code can easily be generated by specifying the appropriate target names during compilation through passing the command line option `-target` to the SAC compiler. Targets are defined in the `sac2crc` configuration file, which presets a large number of parameters such as backend C compiler and linker, various paths to runtime libraries and the SAC standard library, the archiver program, etc, including their parameters. This strategy permits the co-existence of many different compilation paths.

```
target nativephi:
 CC          := "icc"
 CCFLAGS     := "-std=c99 -Wall -O3 "
                "-opt-report-file optreport.txt "
                "-fno-alias -fpic "
                "-fp-model fast=2"
 CCLIBDIR    := "-I$SAC2CBASE/include/ "
                "-I/usr/include"
 LD_DYNAMIC  := "icc -fpic -std=c99 -shared -Wall"
 AR_CREATE   := "xiar crs"
 LIB_VARIANT := "nativephi"

target nativephi_knc::nativephi:
 CCFLAGS     += " -mmic"
 LDDYNAMIC   += " -mmic"
 LIB_VARIANT := "nativephi_knc"

target nativephi_knl::nativephi:
 CCFLAGS     += " -xMIC-AVX512"
 LDDYNAMIC   += " -xMIC-AVX512"
 LIB_VARIANT := "nativephi_knl"
```

Figure 2: Customisation of the SAC compiler for native Xeon Phi execution

Figure 2 shows the specific extensions of the `sac2crc` configuration file for Xeon Phi native compilation. The new target `nativephi` selects the Intel C compiler `icc` for binary code generation and sets the compiler flags accordingly. The further two targets `nativephi_knc` and `nativephi_knl` inherit all settings from the `nativephi` target. One may likewise say that they extend the `nativephi` target. The latter two targets add the architecture-specific flags that make the compiler generate custom code for the KNC and the KNL architectures. With the targets properly defined we can now proceed to compile a SAC program, say `myprog.sac`, as follows:

```
> sac2c -target nativephi_knc -mt \
        -maxthreads 300 \
        -o myprog myprog.sac
```

With the command line parameter `-target nativephi_knc` we select code generation for the Knights Corner architecture. With `-mt` we select the multithreaded backend of the SaC compiler, and with `-maxthreads 300` we set the maximum number of threads to 300 instead of the compiler default of currently 32. This is merely an upper bound; the precise number of threads is determined at program startup. In addition to the binary executable file `myprog` the compilation process generates the `optreport.txt` file that contains useful information about the backend binary code generation, e.g. hints on the success of auto-vectorisation. The above command generates Knights Corner code; Knights Landing code can analogously be created using the target `nativephi_knl`.

Now, all we need to do is to transfer the binary executable to the Xeon Phi, connect to it via `ssh` and start the executable from the command line:

```
> scp myprog mic0:/tmp/
> ssh mic0
> ./tmp/myprog -mt <Nthreads>
```

The number of threads must still be specified through the command line argument `-mt`. The Xeon Phi is represented as `mic0`. The trailing zero indicates the sequence number of the Xeon Phi. This feature allows us to distinguish between multiple Xeon Phis in the same system.

Alternatively, we can make use of Intel's `micnativeloadex` script:

```
> micnativeloadex ./myprog -a "-mt <Nthreads>"
```

The first argument of the script is the executable file. The second argument indicates that the following arguments enclosed in double quotes, are the command line arguments of the executable. The script can be used with the additional `-e` parameter in order to customise parameters of the runtime environment.

The most important such parameter is the environment variable `SINK_LD_LIBRARY_PATH`. It sets the path to the directory that contains essential libraries that have to be linked with the executable. The `micnativeloadex` script actually is used to automate the manual procedure and provide a more user-friendly interface. The runtime system takes care of actions that otherwise should be performed by developers such as the transfer of the required libraries to the coprocessor.

One disadvantage of the native model in SaC is that the execution procedure differs from what SaC programmers are used to. Invoking a separate script to run an application is fairly alien to them. The use of the Xeon Phi is not transparent, but is rather exposed to the user who needs to be aware of the architecture specifics. To ease this a short help message is printed that reminds the user to export the `SINK_LD_LIBRARY_PATH` and to use the `micnativeloadex` script.

The missing part of the puzzle concerns SaC's runtime libraries and the comprehensive SaC standard library. Their importance stems from the fact that many SaC features are actually implemented by these libraries. Accordingly, different SaC implementations for different targets need to have their individual versions of the runtime libraries. Furthermore, users who need to keep several versions of SaC on their system must have all the versions of the libraries that must be able to simultaneously exist in one system.

The `LIB_VARIANT` mechanism is SaC's answer to this form of cross compilation. This is another configuration variable from the `sac2crc` file, see Figure 2. It can uniquely be set for each target and results in properly tagged file names. This solution permits us to maintain different versions of the libraries in the same directories. Consequently, every program that is compiled for a specific target, looks for the corresponding library versions with the proper `LIB_VARIANT` extensions.

Of course, the libraries must be built using the `-mt` compiler option as well as the correct target specification. The SaC standard library build sytems provides a handy construct for cross-compiling the libraries with the use of the `CROSS` parameter.

Hence, compiling the libraries for the `nativephi_knc` target is as simple as:

```
> export SAC2CFLAGS="-mt"
> make CROSS="nativephi_knc"
```

Finally, the only necessary action that has to be taken from the user is to set the `SINK_LD_LIBRARY_PATH` environment variable as in:

```
> export SINK_LD_LIBRARY_PATH=\
  $SAC2CBASE/lib/:$SACBASE/stdlib/shared-libs/
```

4.2 Uniform Offload Model

The central idea of the uniform offload model is to restore transparency of parallel execution from the user's perspective and completely avoid exposing any Xeon Phi specifics. We generate executable code that is started entirely conventionally from the host system's command line. The application then *offloads* itself to the Xeon Phi of choice without bothering the user. Consequently, the generated application partially runs on the host system and partially on the Xeon Phi. In the case of KNL the offload target is a potential KNL node in a cluster containing both standard Xeon and Knights Landing processors.

For the purpose of the offload model we define yet another target in the `sac2crc` configuration file, namely `mtoffloadphi`. The first difference to the `nativephi` targets is that instead of the `-mmic` option for Knights Corner or `-xMIC-AVX512` for Knights Landing we activate the

`-offload-attribute-target=mic`

option of `icc`.

During code generation the SaC compiler decorates every function that is used in an offloaded code region with the `__attribute ((target(MIC)))` or with the `__declspec(target(mic))` annotations. Again, the archive tool for producing static libraries for the offload model is

```
int main( int __argc, char *__argv[])
{
  int SAC_res;
  char *argv0;
  int arg_len[__argc];

  for (int i = 0; i < __argc; i++) {
    arg_len[i] = strlen(__argv[i]);
  }

  for (int i = 0; i < __argc; i++) {
    argv0 = __argv[i];
    int argv0_len = arg_len[i]+1;

    #pragma offload target(mic) \
      in(argv0[0:argv0_len]: alloc_if(1) free_if(0)) \
      nocopy(__argv[0:__argc]: alloc_if(1) free_if(0))
    {
      __argv[i] = argv0;
    }
  }

  #pragma offload target(mic) \
          nocopy(__argv ) \
          nocopy(SAC_commandline_argv)
  {
    SAC_MT_SetupInitial( __argc, __argv, 0, 300);
    SAC_MT_PTH_SetupStandalone( 1);

    SAC_commandline_argc = __argc;
    SAC_commandline_argv = __argv;

    SACf__MAIN__main ( &SAC_res);
  }

  return SAC_res;
}
```

Figure 3: Generated program-independent C code; directives offload the arguments and the computation to a Xeon Phi offload target

`xiar` (instead of standard `ar`) used with the command line parameter `-qoffload-build`. Behind the scenes, compiling a library to be linked with an offload program the compiler silently generates two versions, one for the host and one for the Xeon Phi offload target. Compiling the runtime libraries and the standard library using this target and with multithreading enabled will make the libraries available to the linker.

The format of the `main` function of any SAC-generated C code follows a common pattern, that we must alter for the (uniform) offload model. The result is shown in Figure 3. Special care must be taken of the `argv` vector. The `argv` vector is actually an array of pointers each of which points to a string. Hence, the compiler is not able to know the length of each string at compile time. Our strategy in this case, is to perform multiple offloads, one per argument. As shown in Figure 3, we use a `for`-loop to calculate the length of each argument, which is passed as an argument to the clause that follows the offload directive and indicates the length of the current vector. Each string has a trailing zero character, used as a delimiter. This is the reason why the length of each argument has to be increased by one. Inside the body of each offload region, the offloaded string is copied to the right position of the offload target's copy of the `argv` vector.

After offloading the `__argv` vector, another offload region follows to offload the rest of the code. The function `SAC_MT_SetupInitial` determines the actual number of threads. The `SACf__MAIN__main` function does the actual work, i.e. this is the function generated from the user's `main` function in the SAC program. It executes the sequential code and sets up the environment for each thread. As explained in Section 2.2 whenever program execution reaches a data-parallel with-loop, execution will be shared among all threads.

This strategy offloads the entire program execution to the Xeon Phi. The offloaded functions briefly described here, can be offloaded because they are decorated with the appropriate `__attribute ((target(MIC)))` when compiled with the parameter `-offload-attribute-target=mic`. The same recursively holds for the functions that are called by them, whether they reside in the same compilation unit or in some (standard) library.

4.3 Selective Offload Model

The selective offload model imitates our CUDA-based code generation strategy for GPGPUs [18]. This approach keeps the bulk of program logic on the host system and only selectively offloads individual data-parallel operations to the Xeon Phi. For GPGPUs the hardware restrictions require such a model. For the Knights Corner architecture the low sequential performance motivated this approach. Ignoring the fact that we could have adapted much of the existing GPGPU code generation backend, this approach is by far the most intrusive from a compiler perspective. Among others the compiler must organise the on-demand data transfers between host and device memory. The selective offload model generally incurs frequent data transfers. Using a GPGPU this may be unavoidable, but with the Xeon Phi it would remain to be seen how much communication overhead can be offset by increased sequential performance. This would at best be application-specific.

While we were working on this model the Knights Corner architecture was superseded by the Knights Landing architecture. This paradigm shift in hardware away from the PCIe-based accelerator model towards a fully-fledged CPU architecture in conjunction with a significantly improved sequential runtime performance convinced us not to pursue this approach in terms of active implementation any more.

Even with a standard Xeon and a Knights Landing processor sharing the same memory individual with-loops do not appear to be the right granularity to divide application execution between the two architectures. In practice we often observe a merely negligible sequential code section in between two data-parallel ones. In such scenarios it would be much more beneficial to execute the sequential code section on the Xeon Phi as well. Only with more complex application components that expose significant performance affinity with one or the other architecture this form of heterogeneous computing may again become an interesting option. However, for the time being we have not conducted any analyses to this effect.

5 EXPERIMENTAL EVALUATION

This section summarises the results of our experiments with running SaC programs on Xeon Phi systems with both the first generation Knights Corner and with the second generation Knights Landing microarchitectures.

5.1 Setup and Methodology

The KNC experiments were run condurcted on the *Distributed ASCI Supercomputer* (DAS-4) [2]. Our KNC coprocessor has 60 cores (1.05 GHz) and 8 GB of memory. All KNL experiments were done on the *Cartesius* cluster of the Dutch national supercomputing center SURFsara. Our KNL processor has 64 cores (1.3 GHz) with a maximum memory capacity of 384 GB (depending on the memory type).

Every experiment was run five times, and we make use of the average execution time for further evaluation. The experiments were submitted to the job scheduler of the two clusters in order to guarantee isolation from other simultaneously running applications and, thus, meaningful execution time measurements.

5.2 Benchmarks

We make use of two benchmarks that are fairly common in parallel computing: Gaussian Blur (GB) and Matrix Multiplication (MM). Fig. 4 shows a typical SaC implementation of matrix multiplication. Thanks to extensive high-level code transformation of the SaC compiler generates C code from this specification that is similar to textbook implementations of matrix multiplication.

```
double[.,.] matmul( double[.,.] A, double[.,.] B)
{
  Bt = transpose(B);

  C = with {
        (. <= [i,j] <= .): sum( A[i] * Bt[j]);
      }: genarray( [shape(A)[0], shape(B)[1]]);

  return C;
}
```

Figure 4: SAC implementation of Matrix Multiplication

The Gaussian Blur filter is a special case of the image convolution, which produces an output image with less noise and less detail than the original one. Next to the image to be processed Gaussian Blur takes a filter (or weight) matrix as an argument In our experiments we used the 5×5 filter shown in Figure 5.

```
weights = [[ 1,  4,  7,  4, 1],
           [ 4, 20, 33, 20, 4],
           [ 7, 33, 55, 33, 7],
           [ 4, 20, 33, 20, 4],
           [ 1,  4,  7,  4, 1]];
```

Figure 5: Gaussian Blur filter matrix

We discussed a large variety of convolutionary algorithm implementations in SaC in [13]. Hence, we omit any concrete SaC code her and refer the interested reader to [13]. Again, the SaC compiler generates text book level C encodings from high-level functional SaC specifications.

5.3 Native Execution on Knights Corner

Our first experiment on the Knights Corner architecture is about sequential performance. Although the architecture is solely designed for throughput-oriented parallel computing, these experiments yield an interesting performance baseline and expose some interesting properties of our two benchmarks.

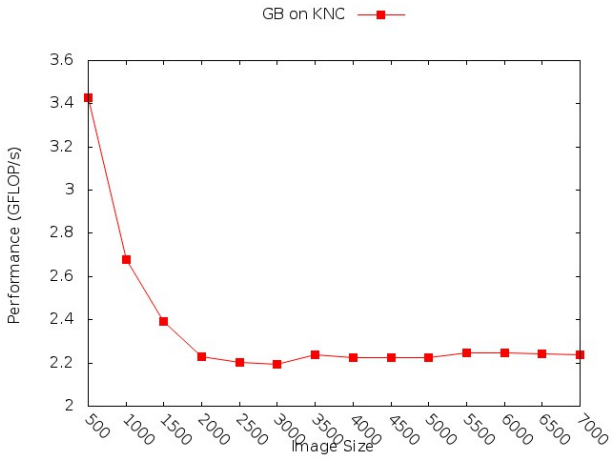

Figure 6: GB: sequential execution

We show results for Gaussian Blur and for Matrix Multiplication in Fig. 6 and Fig. 7, respectively. For both benchmarks we systematically vary the problem size from 500^2 to 7000^2 elements.

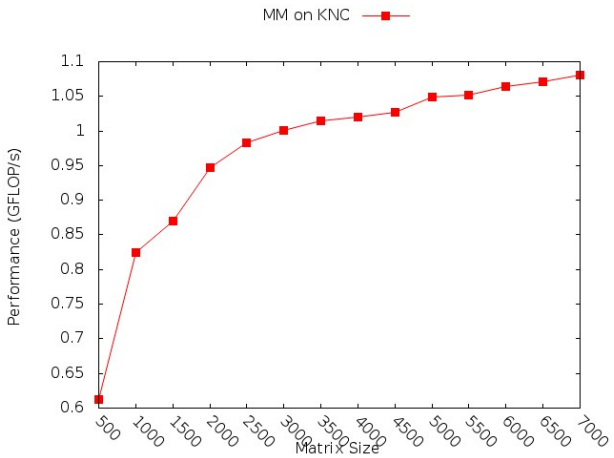

Figure 7: MM: sequential execution

Obviously, we cannot expect to get anywhere close to peak performance with using only a single core. Nonetheless, we achieve a respectible 3.4 GFLOPS with the smallest image size in Gaussian Blur. Increasing the image size reduces performance to about 2.2 GFLOPS. Gaussian Blur is a benchmark that exposes high data locality in the stencil computation. With larger image sizes the cache hit rate decreases.

For Matrix Multiplication we pretty much observe the opposite behaviour with respect to problem size: the smallest problem size yields the worst performance with little more than 600MFLOPS while increasing the problem size allows us to go well beyond 1 GFLOP. In this benchmark data locality is low from the very beginning. Apparently, some overhead increasingly amortises with larger problem sizes.

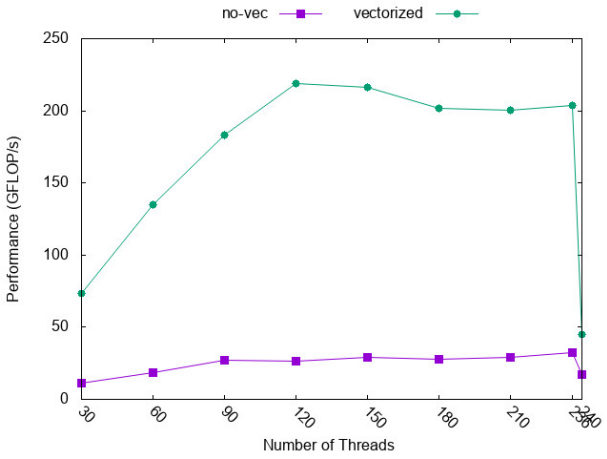

Figure 8: GB: Parallel Performance 5000x5000

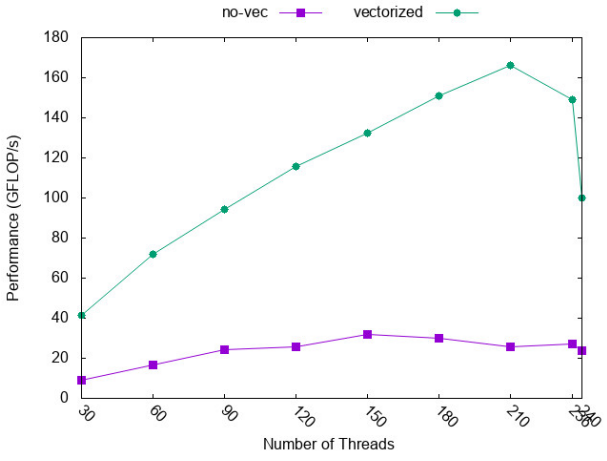

Figure 9: MM: Parallel Performance 3000x3000

Our next round of experiments focusses on parallel performance for a fixed problem size, 5000^2 in the case of Gaussian Blur and 3000^2 in the case of Matrix Multiplication. Figures 8 and 9 show our results.

Now we achieve up to 220 GFLOPS for Gaussian Blur and up to 160 GFLOPS for Matrix Multiplication. In both cases using all 60 cores with 4 threads each cannot be recommended. It is very important to leave one core to the operating system. For both benchmarks our results using 236 threads are considerably better than using all 240 threads supported by the Knights Corner hardware.

Furthermore, it is remarkable that in neither experiment 236 threads yield the best performance. In particular for Gaussian Blur 120 threads already suffice to achieve the best performance. Increasing the number of threads beyond 120 merely results in a performance plateau. This observation can be attributed to the early saturation of the memory subsystem.

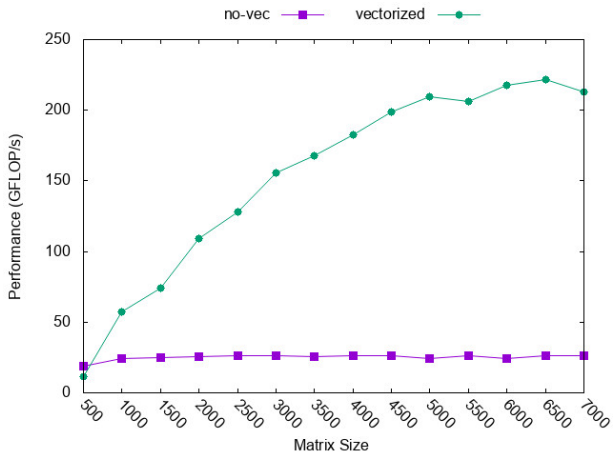

Figure 10: GB: 150 threads vs problem size

Figures 8 and 9 also demonstrate the crucial importance of vectorisation on the Knights Corner architecture. With vectorisation disabled both benchmarks achieve only about 30 GFLOPS maximum performance. Instruction level parallelism exploiting the 512-bit vector registers plays a crucial role to leverage the full potential of the architecture, which comes at no surprise. Positively, we can conclude that the auto-vectoriser of Intel's C compiler does a good job here. In fact, the SaC compiler itself relies on the backend C compiler for machine-targeted vectorisation and only supports this by generating code that in principle should be easily vectorisable in most cases.

Our last round of experiments on the Knights Corner architecture investigates the impact of problem size in parallel execution with a fixed number of threads. We opted for 150 threads and show the results of these experiments in Figure 10 for Gaussian Blur and in Figure 11 for Matrix Multiplication. In contrast to our first experiments, we can observe here that with 150 threads in operation increasing the problem size yields higher performance, also for Gaussian Blur, where we originally observed the opposite behaviour.

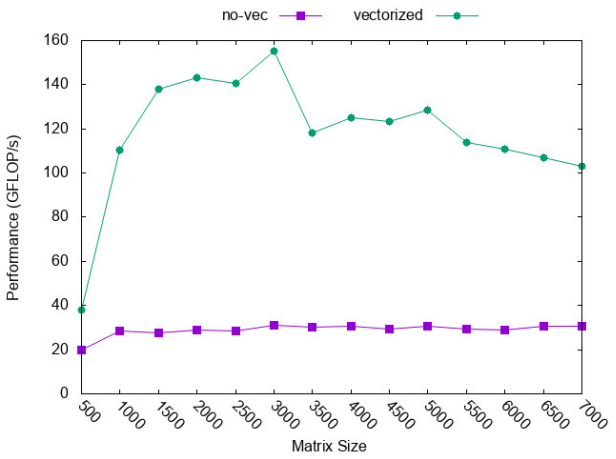

Figure 11: MM: 150 threads vs problem size

These observations are pretty much in line with general experience, namely that certain overheads incurred by parallel execution amortise better over larger problem sizes.

5.4 Native Execution on Knights Landing

Our experiments on the new Knights Landing architecture follow the same strategy as those reported for the Knights Corner architecture. Hence, we start with a glimpse on sequential performance as shown in Figures 12 and 13 for our two running benchmarks.

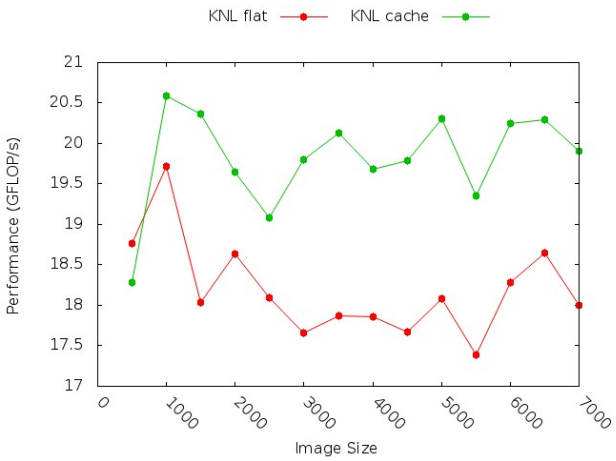

Figure 12: GB: Sequential Performance

What immediately strikes us is the substantially improved sequential performance of the Knights Landing architecture when compared to the results shwon in Figures 6 and 7 for the older Knights Corner architecure. For Gaussian Blur we achieve up to 20 GFLOPS instead of 2.2 GFLOPS for relevant problem sizes. For Matrix Multiplication we still observe a roughly three-fold increase in sequential performance. This allows us the conclusion that depending on the concrete code we can expect a performance increase in sequential execution when going from Knights Corner to Knights Landing that goes well beyond what could be expected from the pure hardware specifications.

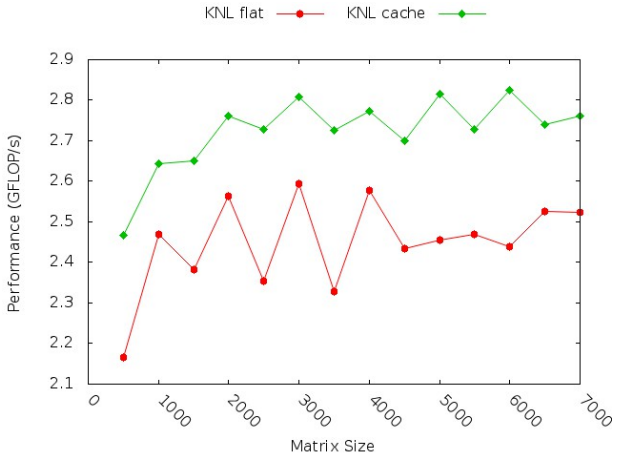

Figure 13: MM: Sequential Performance

From the various memory configuration options of the Knights Landing architecture that we explained in Section 3 we investigate flat mode versus cache mode. In our experiments cache mode is the clear winner across both benchmarks and across all problem sizes.

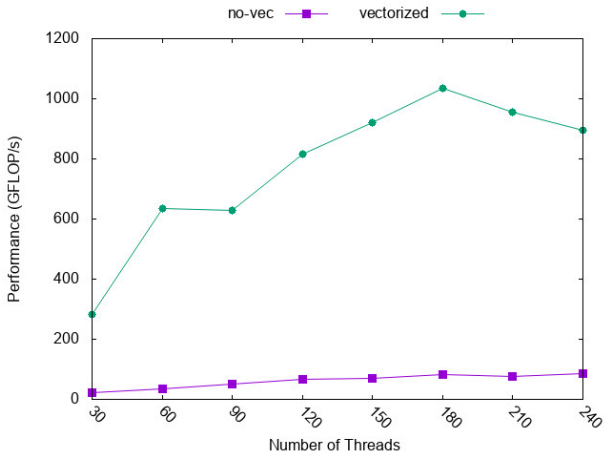

Figure 14: GB: Parallel Performance 5000x5000

Next we show parallel performance for our two benchmarks in Figures 14 and 15. We again go up to 240 threads, but because on Knights Landing this is slightly below the hardware capacity, we do not observe the steep performance drop that was characteristic for the Knights Corner architecture. Just as in the case of Knights Corner, effective utilisation

of the 512-bit vector processing units is essential for performance. And, again, the auto-vectorisation facilities of Intel's C compiler prove sufficient for the job, at least for our two benchmarks, that admittedly do not constitute any particular challenge to this effect.

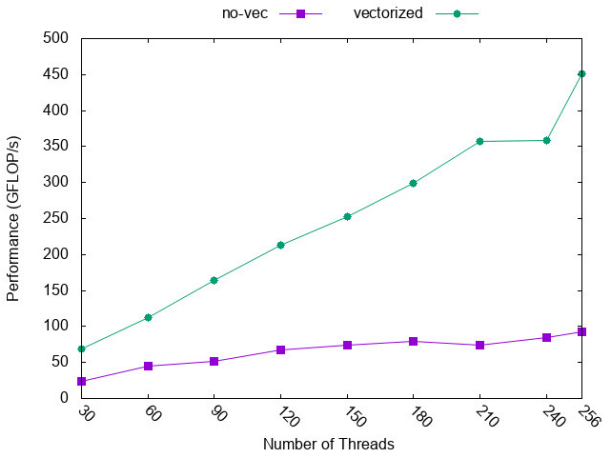

Figure 15: MM: Parallel Performance 3000x3000

In our final round of experiments we again fix the number of threads to 150 and systematically vary the problem size. Otherwise, we use the best options identified so far, namely vectorisation enabled and memory configured in cache mode. Results are shown in Figures 16 and 17. For Gaussian Blur we achieve close to 1 TFLOP in these final experiments. Again, successful vectorisation is crucial for performance.

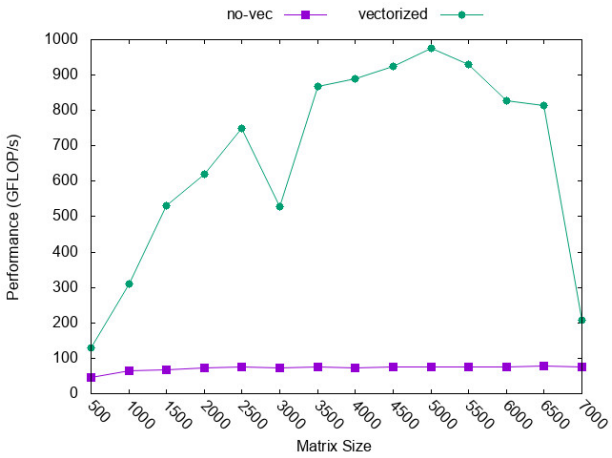

Figure 16: GB: 150 Threads vs Problem Size

5.5 Offload Model Execution

We repeated all experiments with the uniform offload model, as described in Section 4.2. However, we consistently found

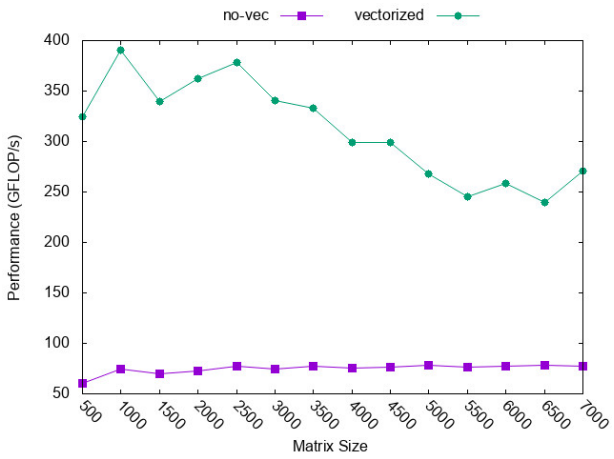

Figure 17: MM: 150 Threads vs Problem Size

merely negligible performance differences between the native model and the uniform offload model. Given that none of our benchmarks does any serious I/O, but both do serious computing on the Xeon Phi this observation is unsurprising. Starting an application on the host system as well as transferring argument data to the accelerator and result data back to the host, both exactly once, merely creates constant overhead that is quickly offset when running non-trivial computations on the Xeon Phi. Hence, for space reasons, we omit further experimental data.

6 RELATED WORK

Among high-level, and in particular among functional programming languages, Intel's Many Integrated Core architecture in general and the Xeon Phi product family in particular have not gained the same popularity as compilation target as GPGPUs have. A notable exception is the imperative data parallel programming language Vector Pascal [5]. Vector Pascal extends ISO Pascal by data-parallel constructs, namely array index ranges.

Recent versions of Glasgow Vector Pascal have adopted quite a few array programming concepts pioneered by SaC [3], and so SaC and Vector Pascal share various aspects in the general approach to data parallel programming.

Major differences are still in the syntactic look-and-feel: C vs Pascal. Moreover, SaC is a purely functional language, whereas Vector Pascal also adopts the imperative paradigm from its host language Pascal. Among others, SaC features completely automatic resource management for memory, cores, etc, whereas Vector Pascal leaves such aspects to the user/programmer.

The Glasgow Vector Pascal compiler was ported to the first generation Xeon Phi, Knights Corner, following a similar strategy as pursued by us [3]. The frontend of the compiler is machine-independent, and, hence, the main challenges addressed lie in the backend, same as for SaC. A major difference between the Glasgow Vector Pascal compiler and our

setting is that they compile all the way down to machine assembly. Abstract machine descriptions serve automatic compiler backend generation for a variety of target microarchitectures. In contrast, we merely compile down to C as a general intermediate representation and make use of Intel's Xeon Phi enabled C compiler `icc` for the final generation of executable code.

The Vector Pascal approach requires explicit incorporation of the advanced vector processing capabilities of the Xeon Phi family into their compiler. In contrast, we can semi-automatically benefit from the auto-vectorisation features of Intel's C compiler as well as the human resources and in-depth hardware information available to Intel's compiler engineers that go way beyond the capabilities of any academic project.

Leaving the realm of high-level programming languages, Fang et al investigated the Knights Corner architecture from a performance engineering perspective [8]. Based on an experimental investigation much more thorough than our's, they confirmed that it is indeed possible to achieve very good runtime performance, not far from Knights Corner's nominal peak performance. The Xeon Phi family's support for well established parallel programming models, like PThreads, OpenMP or MPI, helps programmers to quickly develop initial solutions. Often existing implementations that originally targeted small-scale multi-core systems or networked clusters of workstations can more or less straightforwardly be adapted to run on the Xeon Phi.

However, Fang et al only achieved their high level of performance with a major effort in code/problem analysis, profiling and target-directed, low-level manual tuning. They concluded that the number of applications that might make highly efficient use of Knights Corner may be limited in practice.

7 CONCLUSIONS

We have presented several approaches to target Intel's Xeon Phi processor family, namely the Knights Corner and Knights Landing architectures, from the purely functional array programming language Single Assignment C (SaC). These include running SaC applications natively on the Xeon Phi over offloading entire applications from some host machine to a Xeon Phi processor up to selectively offloading individual data-parallel operations from a host or general-purpose processor to a Xeon Phi Knights Corner or Knights Landing co-processor.

Among others, this adds two more highly interesting and relevant target architectures to SaC's portfolio without sacrificing SaC's fundamental promise, or maybe better goal: compiling a single target- and resource-agnostic source program to a wide variety of architectures without any programmer intervention.

Whereas the third option is only sketched out as a design study, we have indeed realised the other two and ran an extensive set of experiments both on a Knights Corner and on a Knights Landing system. We lost interest in the selective offload model when advancing from Knights Corner to Knights Landing. The first commercial Xeon Phi generation was known for very limited sequential performance. Hence, running sequential parts of an application on a general-purpose host system and only offloading individual data-parallel operations is attractive as it combines the best of two worlds for the benefit of overall application performance. With the very much improved sequential performance of the second generation Knights Landing this motivation looses relevance.

In addition to answering the principle question *Does it work?* two main conclusions can be drawn. Firstly, the multithreaded auto-parallelisation backend of the SaC compiler is adequate for both Xeon Phi architectures. Moreover, the code generated by the SaC compiler appears to work well with the auto-vectoriser of Intel's C compiler `icc`, which is crucial for leveraging the performance potential of the 512-bit vector units of the Xeon Phi family.

Both the native execution model (Section 4.1) and the uniform offload model (Section 4.2) work equally well in practice. Hence, the added "luxury" to effectively hide the parallel compute architecture from the user, other than selecting the appropriate compilation target, seems both affordable and recommended in the SaC setting.

However, hiding the intricacies of parallel computing also has its practical limits. As the experiments in Section 5 consistently demonstrate, the optimal number of threads depends both on the code, on the problem size and, last not least, on the concrete architecture, here KNC or KNL. Making use of all hardware resources often does not lead to optimal performance, not even to mention performance per Watt. Neither is the rule of thumb to leave one or two cores unused sufficient to achieve consistently good performance. Here, the user still needs to run at least some rudimentary experiments to better understand the runtime characteristics of some code on some architecture for the relevant problem sizes.

Comparing our findings with those of Fang et al [8], as discussed in the previous section, we could conclude that our entirely compiler-directed approach, starting from a high-level, functional programming language, does reasonably well. We have not included any of their low-level performance tuning tricks and still obtain useful performance levels. And, this performance comes "for free" from the SaC programmer's perspective.

In contrast, the amount of tuning effort invested by Fang et al in their performance engineering approach would only be justified for a limited number of application kernels. Furthermore, it requires intricate knowledge of the architecture, and with every new generation of hardware the process restarts practically from scratch.

An interesting area of future research will be boards with multiple Knights Landing processors as well as heterogeneous systems consisting of multiple Xeon processors and multiple Knights Landing processors. With the latest developments as of November 2017 and the termination of the Xeon Phi product line by Intel, however, it remains to be seen how our findings can be carried over to future generation compute systems on the way towards exascale computing.

ACKNOWLEDGMENTS

Our sincere thanks go to the Advanced School of Computing and Imaging (ASCI) for providing us with access to a Xeon Phi Knights Corner system as part of their DAS-4 distributed research cluster [2]. Moreover, we thank SURFsara, the Netherlands national supercomputing center, for granting us access to one of their Xeon Phi Knights Landing systems. Last not least, we would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] S. Anthony. 2013. Intel unveils 72-core x86 Knights Landing CPU for exascale supercomputing. (2013).

[2] H.E. Bal, D. Epema, C. de Laat, R. van Nieuwpoort, J. Romein, F.J. Seinstra, C. Snoek, and H. Wijsshoff. 2016. A Medium-scale Distributed System for Computer Science Research: Infrastructure for the Long Term. *IEEE Computer* 49, 5 (2016), 54–63.

[3] M. Chimeh, P. Cockshott, S.B. Oehler, A. Tousimojarad, and T. Xu. 2015. Compiling Vector Pascal to the XeonPhi. *Concurrency and Computation: Practice and Experience* 27 (2015), 5060–5075.

[4] G. Chrysos. 2012. Intel Xeon Phi coprocessor (codename Knights Corner). In *24th IEEE Hot Chips Symposium (HC'12), Stanford, USA*. 1–31.

[5] P. Cockshott. 2002. Vector Pascal Reference Manual. *SIGPLAN Notices* 37, 6 (2002), 59–81.

[6] T. Damkroger. 2017. Unleashing High-Performance Computing Today and Tomorrow. *Intel IT Peer Network* (2017).

[7] M. Diogo and C. Grelck. 2013. Towards Heterogeneous Computing without Heterogeneous Programming. In *Trends in Functional Programming, 13th Symposium, TFP 2012, St.Andrews, UK (Lecture Notes in Computer Science)*, K. Hammond and H.W. Loidl (Eds.), Vol. 7829. Springer, 279–294.

[8] J. Fang, H. Sips, L. Zhang, C. XU, Y. Che, and A.L. Varbanescu. 2014. Test-Driving Intel Xeon Phi. In *5th ACM/SPEC International Conference on Performance Engineering (ICPE'14), Dublin, Ireland*. ACM, 137–148.

[9] M. Feldman. 2017. Intel Dumps Knights Hill, Future of Xeon Phi Product Line Uncertain. *Top500 News* (2017).

[10] P. Gepner. 2015. Intel Architecture and Technology for Future HPC System Building Blocks. In *14th International Symposium on Parallel and Distributed Computing (ISPDC 2015), Limassol, Cyprus*. IEEE.

[11] Clemens Grelck. 2003. A Multithreaded Compiler Backend for High-Level Array Programming. In *2nd International Conference on Parallel and Distributed Computing and Networks (PDCN'03), Innsbruck, Austria*, Mohammed H. Hamza (Ed.). ACTA Press, 478–484.

[12] Clemens Grelck. 2005. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming* 15, 3 (2005), 353–401.

[13] C. Grelck. 2012. Single Assignment C (SAC): High Productivity meets High Performance. In *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary (Lecture Notes in Computer Science)*, V. Zsók, Z. Horváth, and R. Plasmeijer (Eds.), Vol. 7241. Springer, 207–278.

[14] Clemens Grelck, Stephan Herhut, Chris Jesshope, Carl Joslin, Mike Lankamp, Sven-Bodo Scholz, and Alex Shafarenko. 2009. Compiling the Functional Data-Parallel Language SaC for Microgrids of Self-Adaptive Virtual Processors. In *14th Workshop on Compilers for Parallel Computing (CPC'09), IBM Research Center, Zürich, Switzerland*.

[15] Clemens Grelck and Sven-Bodo Scholz. 2006. Merging Compositions of Array Skeletons in SAC. *Journal of Parallel Computing* 32, 7+8 (2006), 507–522.

[16] Clemens Grelck and Sven-Bodo Scholz. 2006. SAC: A Functional Array Language for Efficient Multithreaded Execution. *International Journal of Parallel Programming* 34, 4 (2006), 383–427.

[17] Clemens Grelck and Sven-Bodo Scholz. 2007. SAC: Off-the-Shelf Support for Data-Parallelism on Multicores. In *2nd Workshop on Declarative Aspects of Multicore Programming (DAMP'07), Nice, France*. ACM Press, 25–33.

[18] Jing Guo, Jeyarajan Thiyagalingam, and Sven-Bodo Scholz. 2011. Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In *6th Workshop on Declarative Aspects of Multicore Programming (DAMP'11), Austin, USA*. ACM Press, 15–24.

[19] J. Jeffers and J. Reinders. 2013. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kauffmann.

[20] J. Jeffers, J. Reinders, and A. Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming: Knights Landing edition*. Morgan Kauffmann.

[21] T. Macht and C. Grelck. 2015. SAC Goes Cluster: From functional array programming to distributed memory array processing. In *Tagungsband des 18. Kolloquiums Programmiersprachen und Grundlagen der Programmierung (KPS 2015), Pörtschach am Wörthersee, Austria*, J. Knoop (Ed.). Technical University of Vienna.

[22] T.G. Mattson, R.F. van der Wijngaart, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. 2010. The 48-core SCC processor: the programmer's view. In *Conference on High Performance Computing Networking, Storage and Analysis (SC'10), New Orleans, USA*. IEEE.

[23] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, et al. 2008. Larrabee: A many-core x86 Architecture for Visual Computing. *ACM Transactions on Graphics* 27, 3 (2008).

[24] R. Smith. 2016. Intel Announces Knights Mill: a Xeon Phi for Deep Learning. *AnandTech* (2016).

[25] A. Sodani. 2015. Knights Landing (KNL): 2nd Generation Intel Xeon Phi processor. In *27th IEEE Hot Chips Symposium (HC'15), Cupertino, USA*. 1–24.